SCHOOL OF INFORMATION SCIENCES AND TECHNOLOGY
THE PENNSYLVANIA STATE UNIVERSITY

Herbal Tutorial

(Supports version 0.9b of the Herbal IDE)

Mark A. Cohen and Frank E. Ritter

Technical Report No. ACS 2004-2
9 May 2005

---

mcohen@lhup.edu
frank.ritter@psu.edu

Phone +1 (814) 865-4455 Fax +1 (814) 865-6426

School of IST, 319 IST Building, University Park, PA 16802

Herbal Tutorial

Mark A. Cohen and Frank E. Ritter
mcohen@lhup.edu ritter@ist.psu.edu
School of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16801-3857

ACS #2004-2
9 May 2005

Abstract

To accommodate this wide range of users, and to promote the use of cognitive systems, it is essential that tools such as high-level languages and development environments are created to allow the modeler to focus more on the problem domain, and less on the nuances of a particular architecture. This tutorial introduces an integrated development environment and high-level behavior representation language called Herbal that represents a step towards creating tools to support a wide range of cognitive model users. This tutorial instructs users how to install Herbal, and describes how to develop two models, including a blocks-world model and a dTank model.

Acknowledgements

The research was supported by the Office of Naval Research, contract N000140210021. Comments from Isaac Council, Bill Stevenson, Geoff Morgan, Urmila Kukreja, Tony Kalus, and collectively the students of IST402 have helped improve this manual. Discussions with SoarTech have influenced our thinking; and they provided support for the study of their SAP that gave rise to some of the results during this work. Steve Haynes and Isaac Council helped develop Herbal.

Table of Contents

1.0 What is Herbal? 1

2.0 Installing the Required Software 1

2.1 Install Protégé 1

2.2 Install the RDF Protégé Plug-in 2

2.3 Install Herbal 3

3.0 Learning Protégé 3

4.0 Creating a Blocks World Model 4

4.1 Creating a New Herbal Project (aka a Protégé Project) 4

4.2 Including the Herbal Soar Ontology 6

4.3 Setting the Model Attributes 8

4.4 Classifying Domain Specific Knowledge 8

4.5 Instantiating Domain Specific Knowledge 11

4.6 Creating the Top State 15

4.7 Assigning Initial Working Memory to the Top State 16

4.8 Creating the MoveBlockToTable Operator and Assigning it to the Top State 17

4.9 Creating the MoveBlockToBlock Operator and Assigning it to the Top State 23

4.10 Creating the GoalStateReached Elaboration and Assigning it to the Top State 26

5.0 Compiling the Blocks World Model 29

5.1 Including Custom Code in Your Model 30

6.0 Impasses and Child States 31

6.1 Generating an Operator-Tie Impasse 31

6.2 Creating an Herbal Child State 31

6.3 Creating an Herbal Impasse 32

6.4 Creating Operators that Resolve the Tie 32

7.0 The Herbal dTank Model 33

7.1 Summary of the dTank Model and How it Promotes Reuse 33

7.2 A Summary of Herbal's Documentation Capabilities 36

7.3 How to Execute the Generated Herbal dTank Model 36

7.4 Suggestions for Expanding the Model 36

8.0 Summary 37

1.0 What is Herbal?

Cognitive architectures are useful to a wide variety of users. Computer scientists, psychologists, cognitive scientists, and various domain experts can all use cognitive architectures. Unfortunately, designing, implementing, and using cognitive architectures can be a difficult task considering that the background and expertise of this diverse set of users varies from novice to expert. In addition, the tasks performed by users of such architectures can vary considerably, and can include a wide variety of tasks. Further discussion of this problem of usability is available in (Ritter et al., 2003). To accommodate the wide range of users, and to promote the use of cognitive systems, development environments allow the modeler to focus more on the problem domain, and less on the details of a particular architecture.

This tutorial introduces an integrated development environment called Herbal (Cohen, Ritter, & Haynes, 2005; Morgan, Cohen, Haynes, & Ritter, 2005; Morgan, Ritter, Cohen, Stevenson, & Schenck, 2005)that acts as a first step towards creating development tools that support the wide range of users of cognitive models. With it, users can create models graphically, and have these models compiled into Soar productions. The productions will work as any Soar model does. The structure of the model can be passed to an associated tool that can help display and explain the model, even as the model runs.

This document also starts to describe a programming style for Herbal, in the same way as the Soar manual (Laird, Congdon, & Coulter, 1999) and the Soar Dogma document (Nuxoll & Laird, 2003).

The design of Herbal is based on a study done with SoarTech's SAP about what questions users ask of a model, as well as an analysis of explanation and explanation types from that broad literature (Council, Haynes, & Ritter, 2003; Haynes, Ritter, Council, & Cohen, submitted).

2.0 Installing the Required Software

The Herbal IDE is built upon and leverages several pieces of software. These systems must be installed before you can begin using Herbal. The following section lists the required software, along with installation instructions.

2.1 Install Protégé

The first step towards getting Herbal installed and running is to download and install Protégé(Stanford Medical Informatics, 2004). Protégé is a graphical ontology editor created and maintained by Stanford Medical Informatics. Herbal is thus "programmed" by defining the model in Protégé. The tool is available for free under the Mozilla Public License (Mozilla, 2004), and can be downloaded from protege.stanford.edu.

Because Protégé is written in Java, it will run on any platform that supports a version 1.4.2 or higher Java Virtual Machine (this includes Windows XP, Linux, and Macintosh machines).

When you download the Protégé installation you can choose between an installation that contains a current JVM, and an installation that assumes you already have the correct JVM installed. If you do not already have a version 1.4.2 or higher compliant JVM installed on your machine, be sure to select the Protégé installation that contains the proper JVM. If you have an earlier version of the JVM, you should uninstall it before continuing with the Protégé install.

From the Protégé download page, you should choose to download the full Protégé 3.0 installation. Protégé 3.0 is available for several different platforms including Windows, Mac OSX, and Linux. Finally, remember to select the package that contains the Java virtual machine if you do not currently have a version 1.4.2 compliant JVM installed.

After the download completes, double click the downloaded file and follow the installation instructions on the screen as they appear during the installation.

2.2 Install the RDF Protégé Plug-in

Herbal uses the Resource Description Framework (RDF) as its main representation language, and Herbal relies on Protégé to generate RDF when a project is saved (W3C, 2004). As a result, the Protégé backend plug-in that generates RDF must be installed before you can start using Herbal. Because this plug-in is part of the full Protégé distribution, it is installed by default when the full version of Protégé is installed.

To test Protégé and to test that the RDF plug-in was correctly installed by the Protégé installation program, run Protégé by double clicking on its icon, and then look for "RDF Files" listed as a project format option in the Protégé open project dialog (which comes up as Protégé starts) as shown in Figure 1. If you do not see "RDF Files" listed as a project format, it is likely that you did not download the full version of Protégé.

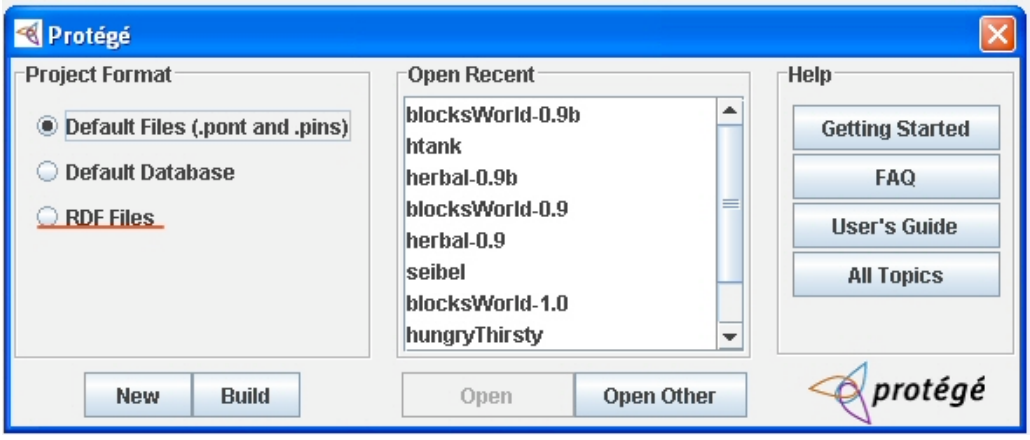

Figure 1. Testing the RDF plug-in installation. If "RDF Files" appears in this startup screen (middle, left), RDF support has been successfully installed.

2.3 Install Herbal

The latest Herbal distribution file can be downloaded from acs.ist.psu.edu/projects/herbal/. Installing the Herbal IDE requires: (a) unzipping the Herbal distribution folder and (b) placing the herbalWidgets.jar file into the Protégé "plugins" folder.

(a) To install Herbal, create a folder on your local hard drive called herbal-X, where X is the version number. Unzip the herbal-X distribution into this folder. When the decompression process is complete, browse to the herbal-X folder and be sure that the files listed in Table 1 are located in this folder.

Table 1. Files in the Herbal Distribution (X represents the version number). These files can be found in your Herbal installation folder.

Herbal Protégé Files
herbal-X.pprj
herbal-X.rdf
herbal-X.rdfs
herbalWidgets.jar
Herbal Compiler Files
herbal-X.xslt
Blocks World Example Files
blocksWorld-X.pprj
blocksWorld-X.rdf
blocksWorld-X.rdfs

(b) To complete the installation move herbalWidgets.jar into the Protégé "plugins" folder. The Protégé "plugins" folder is located inside the Protégé installation folder (typically Protege_VERSION\plugins). You must restart Protégé for these changes to take effect.

3.0 Learning Protégé

In order to use Herbal, it is essential that you have a basic understanding of Protégé because Herbal uses Protégé for its interface. The best way to learn Protégé is to follow the Protégé tutorial. To start the tutorial, run Protégé and click on the "Getting Started" button located on the Protégé open project dialog (Figure 2). This will launch a browser containing the Protégé tutorial. You can also get to the tutorial by going directly to protege.stanford.edu/doc/tutorial/get_started/. Note, the current (3.0) Protégé tutorial uses screen shots from Protégé 2.

After completing the Protégé tutorial, it is a good idea to keep the Protégé user's guide (Stanford Medical Informatics, 2003) close by. The user's guide can be accessed by running Protégé and clicking on the "User's Guide" button located on the Protégé open project dialog (Figure 2 shows how). Alternatively, you can access the user's guide online at protege.stanford.edu/doc/users_guide/.

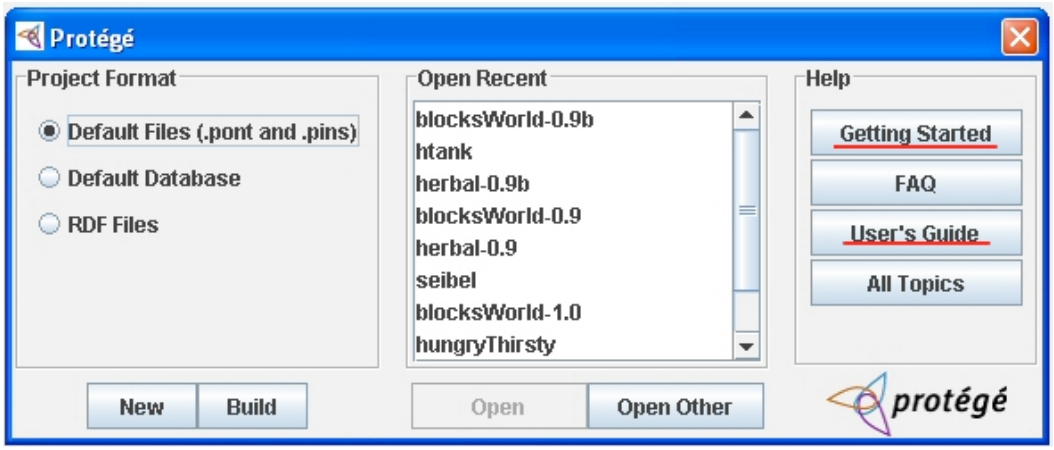

Figure 2. Getting started with Protégé.

It is important that the Protégé tutorial is completed before the reader attempts to create Soar models using Herbal. (You can skip the query part, however.) Also, Macintosh users should note that "option" is the control key.

4.0 Creating a Blocks World Model

This section explains how to create a simple Soar model using Herbal. The model that will be created is a simple solution to the blocks world problem presented in the Soar User's Guide (Laird et al., 1999) and included in most Soar distributions. This problem involves stacking three blocks on top of each other. The initial configuration has three blocks, labeled A, B, and C, sitting on a table. The goal is to stack block A on top of block B; block B on top of block C; and block C should be on the table.

4.1 Creating a New Herbal Project (aka a Protégé Project)

The first step in creating a model using the Herbal IDE is to run Protégé and create a new RDF Schema based project. This can be accomplished by performing the following steps:

1. Create a new folder called BlocksTutorial and copy the herbal-X.xslt, herbal-X.pprj, herbal-X.rdf, and herbal-X.rdfs files from your Herbal installation folder into this new folder (again, X represents the Herbal version number). This step needs to be done whenever you create a new model.

Installation Tip: *For your convenience, the BlockTutorial folder is created for you and included in the Herbal distribution.*

2. Run Protégé. How this is accomplished may differ depending on your platform. On Windows this is accomplished by double clicking on the Protégé icon. See the Protégé documentation if you are having problems starting Protégé.

3. You will be presented with a Protégé dialog that allows you to create new projects, or open an existing project. In this dialog select "RDF Files" and then click on New to create your Herbal project (as shown in Figure 3).

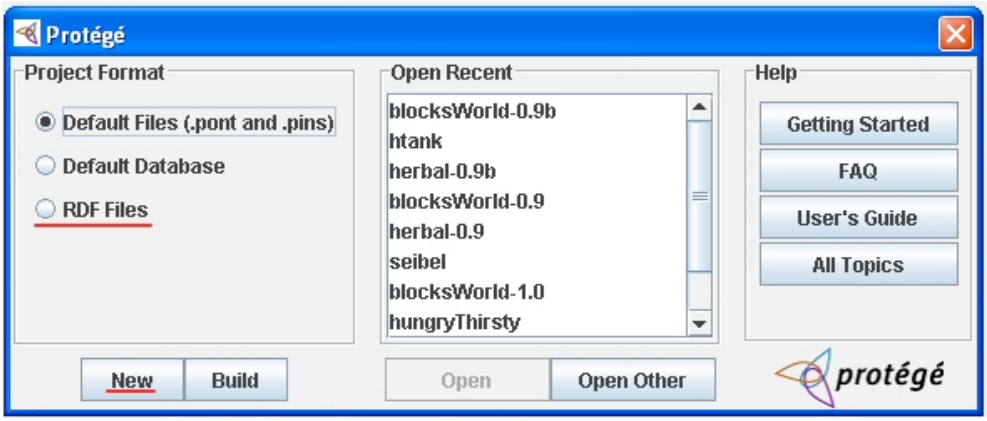

Figure 3. The New Project Dialog.

4. After the new project has been created save the project in the "BlocksTutorial" folder you created in Step 1. To do this select "Save As…" from the "Project" menu and then click on the button next to the Project field in the "RDF Files" dialog box (as shown in Figure 4).

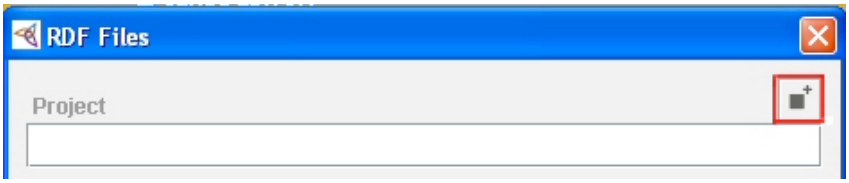

Figure 4. The Browse button in the Save As RDF Files Dialog.

Clicking on this button will display a file "Save As" dialog box. Browse to the "BlocksTutorial" folder you created in Step 1, **type the name of your new project ("BlocksTutorial") in the file name edit box,** and then click on the select button. Take the defaults provided by Protégé for the "Classes file name" and the "Instances file name", and enter "http://protege.stanford.edu/kb#blocks" for the Namespace field. When you are done, the dialog should looks something like Figure 5.

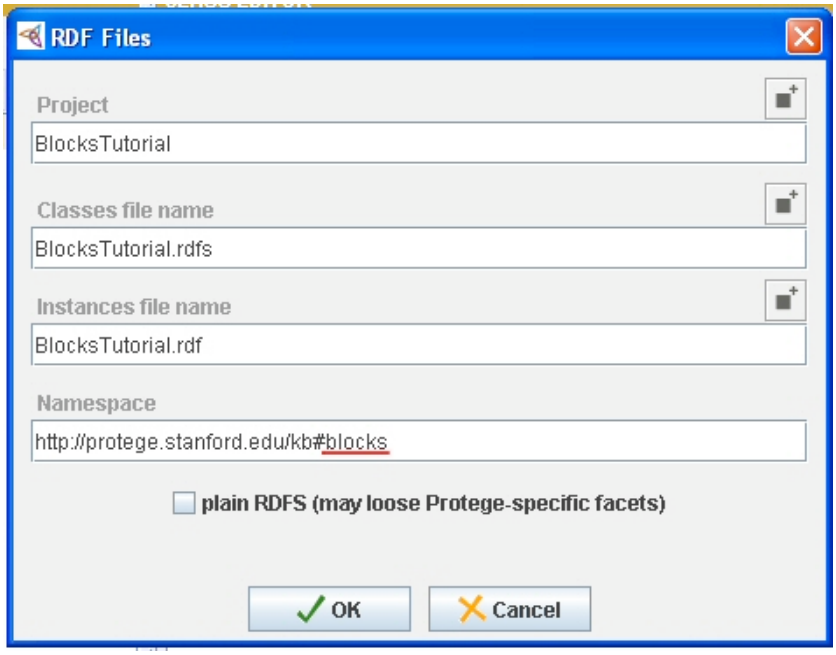

Figure 5. RDF Files Save As dialog.

Protégé Tip: *There is a bug in Protégé that causes the "Save As" dialog box to behave differently than may be expected. If you find that you cannot dismiss the "Save As" dialog box, you may have forgotten to enter the name that you want to give to your new project file in the file name edit box. Repeat step 4 and pay close attention to the text in bold.*

5. Click "OK" and the BlocksTutorial project, consisting of three files (BlocksTutorial.pprj, BlocksTutorial.rdfs, and BlocksTutorial.rdf), will be saved in your "BlocksTutorial" folder (pprj stands for Protégé project).

Protégé Tip: *Each new project consists of three related files: the pprj file, the rdf file, and the rdfs file. The pprj file is the Protégé project file. The rdf and rdfs files contain your model's definition in RDF format. These files, along with the Herbal pprj, rdf, rdfs, and xslt (the compiler definition file) files must remain in the same directory. Do not copy or otherwise move these files to different locations or else Protégé will have trouble finding them later. In addition, all of these files are required by the project, so if you wish to backup, move, or email a project someone else, you need to include all of the files in your project directory.*

4.2 Including the Herbal Soar Ontology

Before you can start creating knowledge, states, operators and elaborations, you must include the Herbal Soar Ontology. This ontology was also created using Protégé (and the RDF plug-in), and is defined in the following three files: herbal-X.pprj, herbal-X.rdf, and herbal-X.rdfs. These files will be located in your "BlocksTutorial" folder.

The Herbal Soar Ontology should be included in your project by performing the following steps:

1. From the "Project" menu, select the "Include Project…" menu item. This will present you with the "Included Project" dialog box.

2. From the "Included Project" dialog box select "herbal-X.pprj" and click on "Open". A dialog box that reads "Changing the included projects will cause the current project to be saved and reloaded" will appear; click on "OK" to complete the include process.

3. When the process completes, you will be presented with the "Namespaces" dialog box. Namespaces are commonly used in XML to help uniquely define elements that make up an XML document. Because RDF is XML-based, Protégé uses namespaces to distinguish between all of the RDF projects currently used by your Protégé project. Notice that the dialog shown in Figure 6 lists four namespaces: the default namespace for your "BlocksTutorial" model, the "herbal" namespace for the Soar ontology, and the remaining two namespaces for RDF. Verify that your namespaces dialog looks like Figure 6.

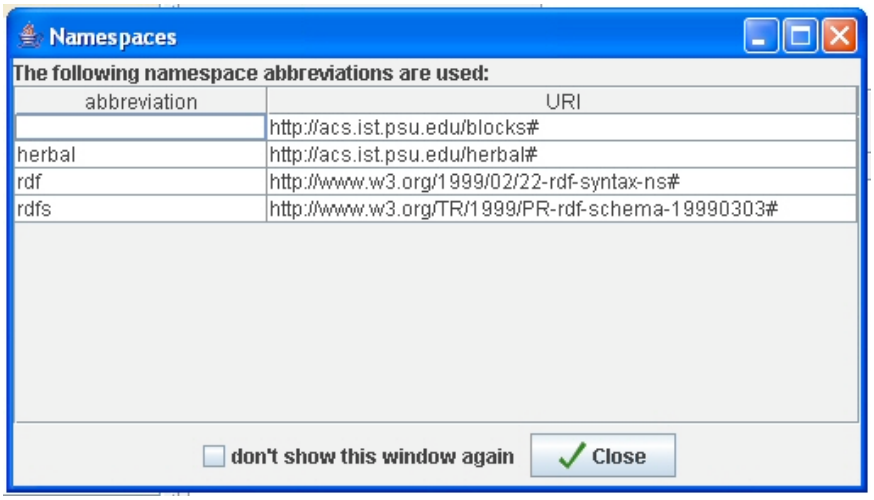

Figure 6. Namespaces dialog.

4. Dismiss the "Namespaces" dialog by clicking on the "Close" button. This completes the import process of the Herbal ontology, and you are now ready to start creating your Soar model.

For Advanced Herbal Users: *The same process can be used to include any RDF-based Protégé project. In other words, you can reuse pieces of other Soar models created in Herbal (i.e., working memory elements, states, operators, elaborations, actions, and conditions) by just including them into your next project in this fashion.*

4.3 Setting the Model Attributes

The first step towards defining the model is to specify the top level model attributes. These attributes are important because they provide documentation that can help explain the purpose of the model, and how the model works. Setting these attributes can be accomplished using the "Model Attributes" tab. This tab is unique to Herbal, and will only be displayed if the "herbalWidgets.jar" file was copied to the Protégé "plugins" folder (as described in the section entitled "Installing the Herbal Files"), and if Protégé was configured to use the plug-in as described below.

To configure Protégé to use the "Model Attributes" tab, from the Project menu select the "Configure…" menu item. This will display the Configure dialog box. In this dialog box select the "Tab Widgets" tab and then check the "ModelAttributesTab" check box. If you do not see the "ModelAttributesTab" listed here, please close down Protégé, make sure the "herbalWidgets.jar" file is located in the Protégé "plugins" folder, and then restart Protégé.

After checking the "ModelAttributesTab" check box, click on the OK button and you should see the tab appear as the last tab in the main Protégé window within a few seconds (Figure 7). If you still don't see the "ModelAttributesTab", restart Protégé.

The fields located on the "Model Attributes" tab should be populated before the rest of the model is defined. Take the time now to populate these fields similar to what is shown in Figure 7.

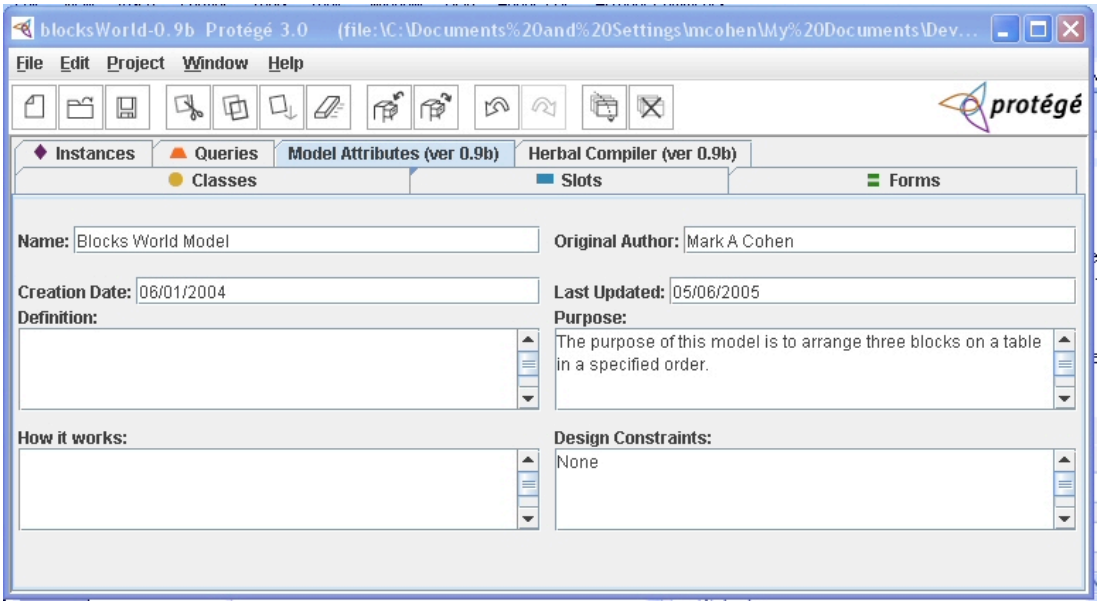

Figure 7. The model attributes tab.

4.4 Classifying Domain Specific Knowledge

The next step in the model creation process is to define and instantiate model knowledge. For this particular model, we are going to define three different classes of knowledge: Block,

Table, and OnTop. These classes, and their attributes, come directly from the Soar User's Guide (Laird et al., 1999). The first two classes are self-explanatory; the third, OnTop, will be used to specify what a block is physically on top of. To define these three classes, perform the following steps:

1. Make sure the "Classes" tab is selected and expand the "herbal:HerbalClass" node in the class tree (Figure 8). These classes are part of the Herbal ontology, and are often preceded by "herbal:" namespace. The Herbal compiler will often use this namespace in the generated Soar code.

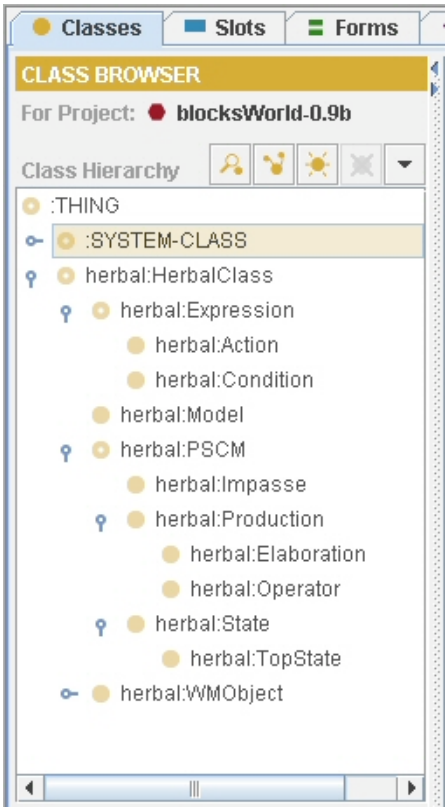

Figure 8. The Herbal class hierarchy.

2. Creating new classes is a typical Protégé task and was described in the Protégé tutorial. At this point you should be familiar with how to create classes. All domain specific classes must be subclasses of the "herbal:WMObject" class, so be sure to create the Block, Table, and OnTop relationship as subclasses of "herbal:WMObject". Don't worry about assigning attributes to these classes until the next step. When you are done creating the classes, the class hierarchy should look Figure 9.

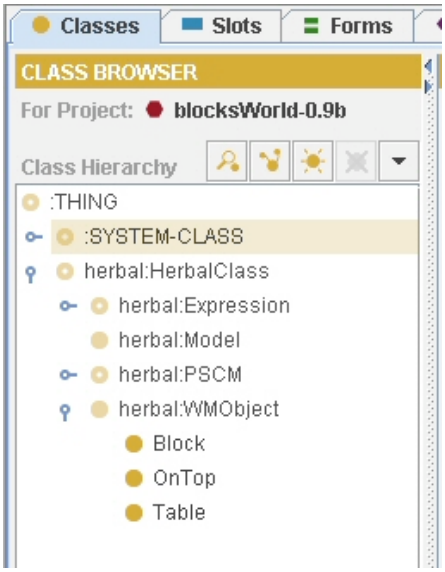

Figure 9. Creating new working memory objects.

3. The next step is to assign attributes to these new classes. You should be familiar with the task of assigning attributes from the Protégé tutorial. Because these new classes are subclasses of "herbal:WMObject", each class will automatically inherit the "herbal:Name", "herbal:Definition", "herbal:How-it-works", and "herbal:Purpose" attributes. However, you will need to add the attributes shown in Figures 10-12 to the Block, Table, and OnTop classes.

Protégé Tip: *Protégé does not allow you to create two attributes with the same name. As a result, classes that have similar attributes will be sharing the attributes they have in common. To accomplish this, you must use the create button to create an attribute the first time. However, from that point on you can reuse the attribute with other classes by clicking on the add button. For example, both the Block class and the Table class require an attribute named "Clear". You only have to create that attribute once using the create button, and then reuse it in the Table class by using the add button.*

Block:

Template Slots

Name	Cardinality	Type
Clear	single	String
herbal:Definition	required single	String
herbal:How-it-works	single	String
herbal:Name	required single	String
herbal:Purpose	single	String
Type	single	String

Figure 10. Block attributes.

Table:

Template Slots

Name	Cardinality	Type
Clear	single	String
Height	single	Integer
herbal:Definition	required single	String
herbal:How-it-works	single	String
herbal:Name	required single	String
herbal:Purpose	single	String
Type	single	String

Figure 11. Table attributes.

OnTop:

Template Slots

Name	Cardinality	Type
Bottom	single	Instance of Block or Table
herbal:Definition	required single	String
herbal:How-it-works	single	String
herbal:Name	required single	String
herbal:Purpose	single	String
Top	single	Instance of Block

Figure 12. OnTop relationship attributes.

4.5 Instantiating Domain Specific Knowledge

Protégé Tip: *Due to a bug in Protégé, some of the data entry forms that you will be using to enter knowledge will not be laid out cleanly. Fortunately, Protégé allows you to customize these data entry forms. To accomplish this, click on the "Forms" tab in Protégé and arrange the forms for the Block, Table, and OnTop classes. You can arrange these forms in Protégé easily by just dragging the data entry controls to a location on the screen that will make for easy data entry.*

Be creative and try and create forms that will be easy to use! You may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry (i.e., the Definition field). If you forget to do this you will only be allowed to enter a single line of text in these fields! If you have further questions about how to customize data entry forms in Protégé please see the Protégé documentation.

Once the domain specific classes have been created, you are ready to instantiate knowledge. For the Blocks World problem the knowledge consists of three blocks, all of which have nothing on top of them. In addition, there is one table, and three OnTop relationships. Each relationship specifies that a block is on the table. In other words, the environment begins with three blocks sitting on a table. To instantiate this knowledge:

1. Make sure the "Instances" tab is selected and expand the "herbal:HerbalClass" node in the instances tree.

2. Creating instances of a class is a typical Protégé task and was described in the Protégé tutorial. At this point you should be familiar with how to create instances using Protégé (working from Instances tab, select the class you want to instantiate and click on the create button).

Protégé Tip: *When creating instances of classes from the Protégé Instances tab, be sure to set the display slot for the class you are going to be using. The display slot specifies what attribute of your instance will be used as a label when Protégé needs to list your object. If you see labels like "blocksWorld-Instance_30000" it means you forgot to set the display slot!*

For the blocks world, you want to create three blocks, one table, and three OnTop relationships. Tables 2-8 specify the attribute values for each of the objects you will be creating (you may also choose to fill in the "herbal:Purpose" and "herbal:How-it-works" fields for your objects, if you wish).

Protégé Tip: *In the tables below, * denotes that the attribute is actually a reference to another object. For example, the OnTop relationship contains the Bottom and Top attribute that actually reference an instance of a block or a table. Using the add button in Protégé you can select existing instances in order to populate these attributes.*

Protégé Tip: *When creating new attributes for a class, you may want to stick to a convention of always starting your attributes with a capital letter. This is because Protégé will capitalize the first letter in attribute names automatically when displayed on a Protégé form. This will help prevent confusion and prevent errors caused by incorrect case. For the value, true and false in particular, they should be lower case.*

Table 2. Attributes for block A.

Herbal:Name	A
Herbal:Definition	A block with the letter A printed on it
Clear	true
Type	Block

Table 3. Attributes for block B.

Herbal:Name	B
Herbal:Definition	A block with the letter B printed on it
Clear	true
Type	Block

Table 4. Attributes for block C.

Herbal:Name	C
Herbal:Definition	A block with the letter C printed on it
Clear	true
Type	Block

Table 5: Attributes for the table object.

Herbal:Name	Table
Herbal:Definition	A table
Clear	true
Type	Table
Height	10

Table 6. Attributes for the AOnTopOf relationship object.

Herbal:Name	AOnTopOf
Herbal:Definition	A relationship that specifies what A is on top of
Bottom	Table*
Top	A*

Table 7. Attributes for the BOnTopOf relationship object.

Herbal:Name	BOnTopOf
Herbal:Definition	A relationship that specifies what B is on top of
Bottom	Table*
Top	B*

Table 8. Attributes for the COnTopOf relationship object.

Herbal:Name	COnTopOf
Herbal:Definition	A relationship that specifies what C is on top of
Bottom	Table*
Top	C*

3. When you are done instantiating your knowledge, your instance hierarchy should resemble Figures 13, 14, and 15.

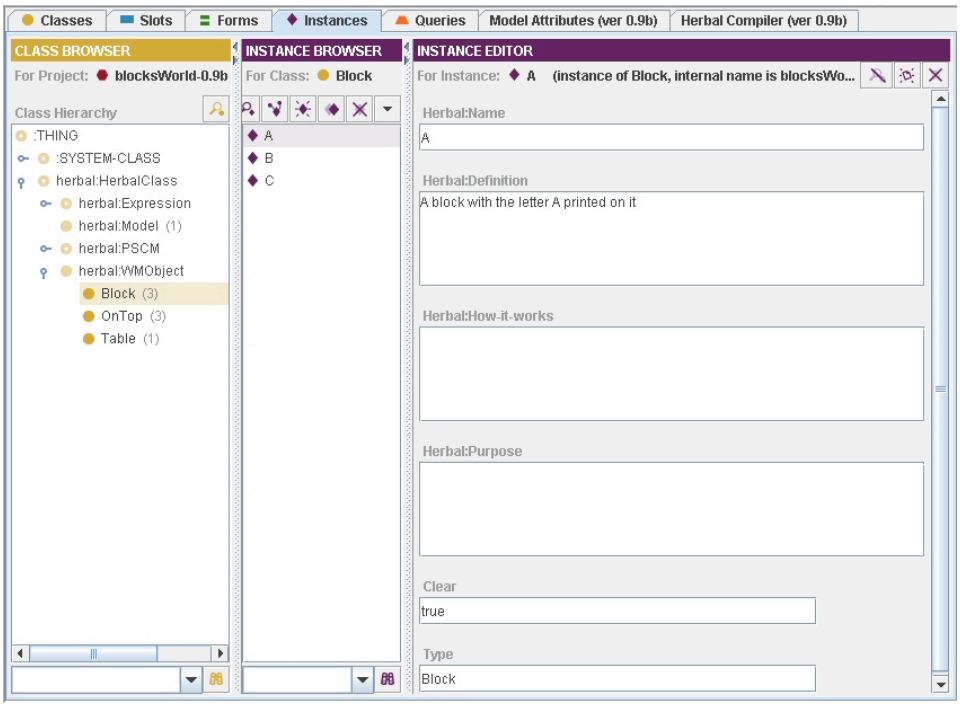

Figure 13. Block instances.

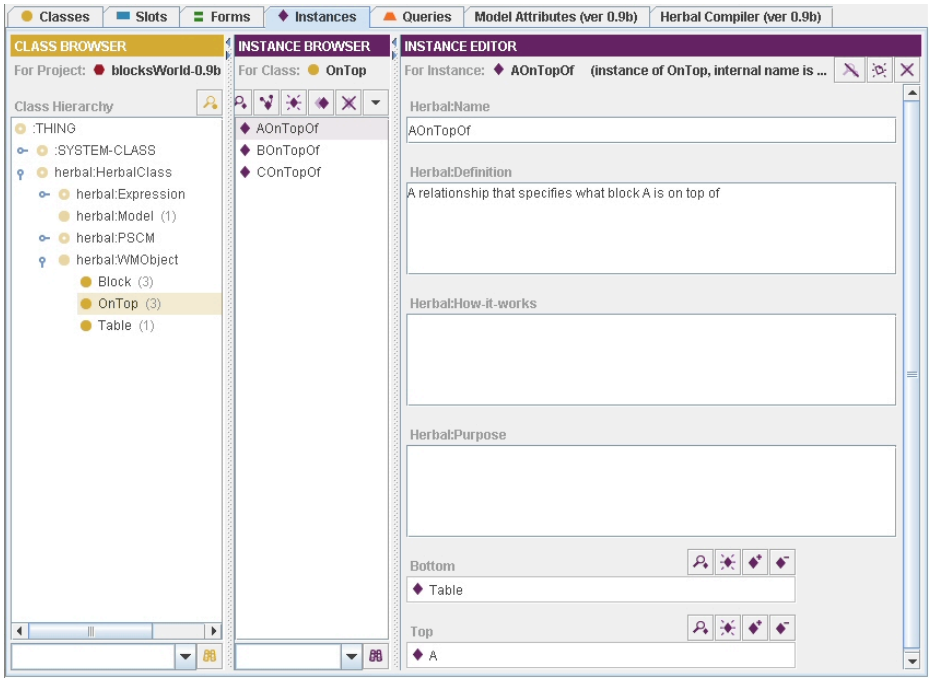

Figure 14. OnTop instances.

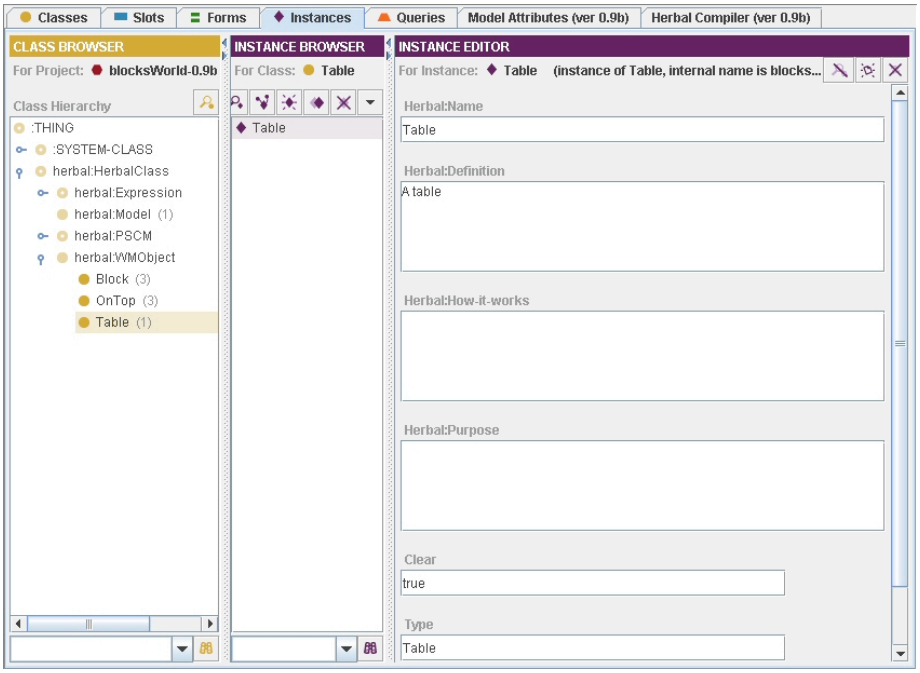

Figure 15. Table instance.

4.6 Creating the Top State

With the domain knowledge classified and instantiated, it is possible to start defining the behavior of the model. This is accomplished by instantiating objects based on the Soar ontology that was imported into the project earlier. The first Soar object that should be created is the top state (currently Herbal only supports the creation of a single, top level state).

1. Make sure the "Instances" tab is selected and then expand the "herbal:HerbalClass" node, then herbal:PSCM, and then the "herbal:State" node in the instances tree. You should see an "herbal:TopState" class located as a subclass of the "herbal:State" class.

2. For the blocks world model, you want to create a single instance of the "herbal:TopState" class named "BlocksWorldState". This state has several important attributes: a name and a definition, as well as a list of operators, elaborations, and working memory.

> ***Protégé Tip:*** *When creating instances of classes from the Protégé Instances tab, be sure to set the display slot for the class you are going to be using. The display slot specifies what attribute of your instance will be used as a label when Protégé needs to list your object. If you see labels like "blocksWorld-Instance_30000" it means you forgot to set the display slot!*

3. After creating the top state object, give the state a name ("BlocksWorldState") and a definition ("The Blocks World Problem Space").

   Once again, you may need to customize the form used for editing the top state. You

can use the Forms tab in Protégé to arrange the display of the form to your liking. The right hand side of Figure 15, for example, can be modified to display the "Type" above the "Clear", or to display the "Description: last.

Protégé Tip: *Once again, you may wish or need to arrange these forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

4.7 Assigning Initial Working Memory to the Top State

Creating the rest of the blocks world model consists of filling in the attributes of the top state. This includes the state's initial working memory, elaborations, and operators. To add initial working memory to the top state, follow these steps:

Protégé Tip: *Once again, when filling in the attributes you may need to arrange the forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

1. All of the instances of domain knowledge created earlier are possible candidates for a state's initial working memory. For the blocks world model, all three blocks, the table, and the three OnTop relationships should be included in the initial working memory for the top state. Adding these working memory elements is a simple task, just click on the plus sign next to the "Herbal:WorkingMemory" attribute and select the working memory instances that have already been created. This is a commonly forgotten task, so be sure to add your working memory to the state.

2. When you are done, the working memory attribute for the top state should resemble Figure 16.

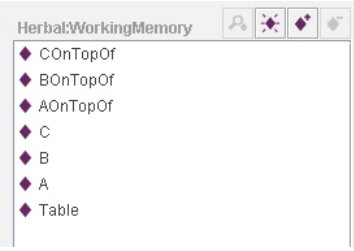

Figure 16. Top state working memory.

As shown in Listing 1, adding these working memory elements to the top state will create an elaboration in the generated Soar code that attaches the appropriate attributes to the top state.

```
sp {elaborate*initial-state*BlocksWorldState
   (state <BlocksWorldState> ^type state ^superstate nil)
   -->
   (<BlocksWorldState>
      ^herbal:Definition |The Blocks World Problem Space|
      ^herbal:Name |BlocksWorldState|
      ^herbal:WorkingMemory <Table>
      ^herbal:WorkingMemory <AOnTopOf>
      ^herbal:WorkingMemory <BOnTopOf>
      ^herbal:WorkingMemory <COnTopOf>
      ^herbal:Elaborations <GoalStateReached>
      ^herbal:WorkingMemory <A>
      ^herbal:WorkingMemory <B>
      ^herbal:WorkingMemory <C>   )
   (<Table>
      ^blocks:Clear |true|
      ^blocks:Height  22
      ^blocks:Type |Table|
      ^herbal:Definition |A table|
      ^herbal:Name |Table|   )
   (<AOnTopOf>
      ^herbal:Definition |A relationship that specifies block A is on top of|
      ^herbal:Name |AOnTopOf|
      ^blocks:Bottom <Table>
      ^blocks:Top <A>   )
   (<A>
      ^blocks:Clear |true|
      ^blocks:Type |Block|
      ^herbal:Definition |A block with the letter A printed on it|
      ^herbal:Name |A|   )
   (<BOnTopOf>
      ^herbal:Definition |A relationship that specifies block B is on top of|
      ^herbal:Name |BOnTopOf|
      ^blocks:Bottom <Table>
      ^blocks:Top <B>   )
   (<B>
      ^blocks:Clear |true|
      ^blocks:Type |Block|
      ^herbal:Definition |A block with the letter B printed on it|
      ^herbal:Name |B|
   (<COnTopOf>
      ^herbal:Definition |A relationship that specifies block C is on top of|
      ^herbal:Name |COnTopOf|
      ^blocks:Bottom <Table>
      ^blocks:Top <C>   )
   (<C>
      ^blocks:Clear |true|
      ^blocks:Type |Block|
      ^herbal:Definition |A block with the letter C printed on it|
      ^herbal:Name |C|   )  }
```

Listing 1. Soar code created by Herbal showing how working memory elements are attached to the top state.

4.8 Creating the MoveBlockToTable Operator and Assigning it to the Top State

The simple blocks world solution that will be implemented in this tutorial consists of two operators and one elaboration. The two operators are “MoveBlockToTable” and “MoveBlockToBlock”. The “MoveBlockToTable” operator checks the top state to see if there is a clear block that is currently not on the table, and if there is a clear spot on the table. If these conditions are satisfied, the block is moved to the table.

The "MoveBlockToBlock" operator checks the top state to see if there are two clear blocks, and that the first clear block is not on top of the second clear block. If these conditions are satisfied, the first block is put on top of the second block.

Creating these two operators in Herbal consists of instantiating two "herbal:Operator" objects and assigning them to the top state. The following steps describe this process:

1. Make sure the "Instances" tab is selected and expand the "herbal:HerbalClass" node, and then the "herbal:Production" node in the instances tree.

2. Create a new "herbal:Operator" object and assign the following attributes:

   Herbal:Name: MoveBlockToTable
   Herbal:Definition: Moves a clear block onto a clear table
   Herbal:Preference: Indifferent

3. Once the operator object is created, the next step is to create and assign the conditions that will cause the operator to be proposed. This is done by instantiating "herbal:Condition" objects. Each "herbal:Operator" object can have one or more conditions assigned to it. If more than one condition is assigned to an operator, the conditions are combined using a logical AND. Because the "MoveBlockToTable" operator should be proposed when a clear block and a clear table exists, we need to create a condition object that matches this situation. To create a new condition object, click on the create button (shown below in Figure 17) next to the operators "Herbal:Conditions" attribute:

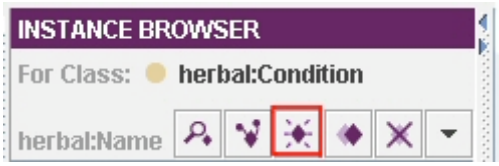

Figure 17. Creating a new condition.

4. When the new condition dialog box is displayed you should assign the new condition the attributes shown in Figure 18.

Protégé Tip: *Once again, when filling in the attributes you may need to arrange the forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

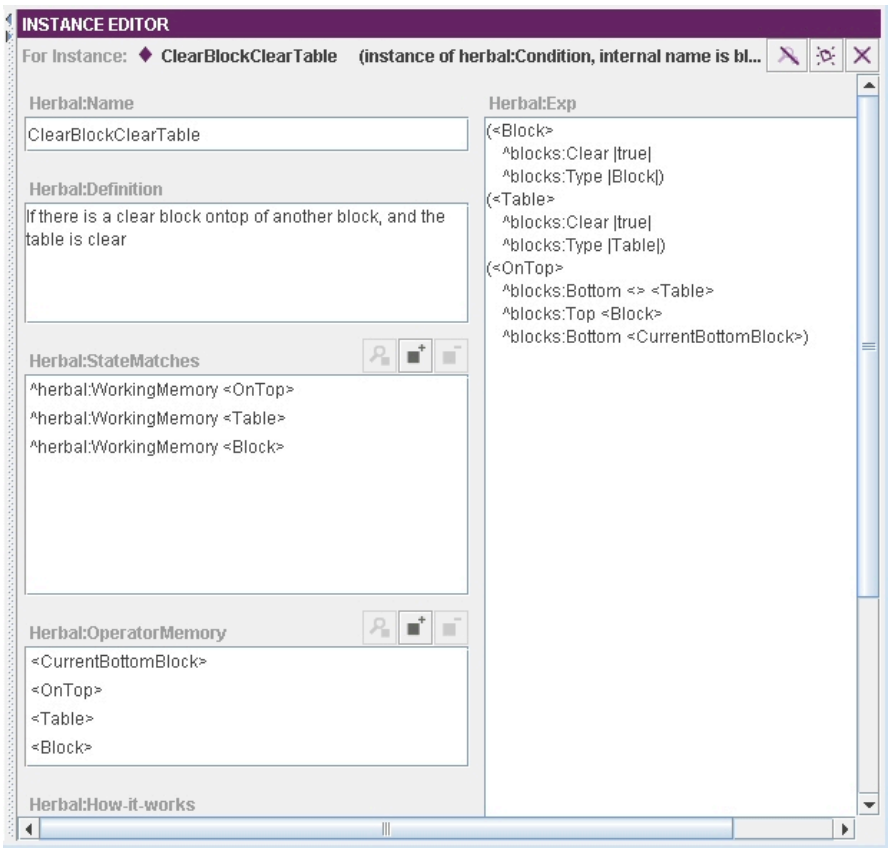

Figure 18. ClearBlockClearTable condition.

To better understand how a condition object works, the attributes in this dialog box should be explained in more detail. The "herbal:Name" and "herbal:Definition" are self explanatory, so we should start the "herbal:StateMatches" attribute. The "herbal:StateMatches" attribute allows the model designer to specify what objects should be searched for as working memory elements on the state associated with the condition. Because this condition will be assigned to an operator on the top state, the top state will be searched for working memory elements that match the objects listed in the "Herbal:StateMatches" attribute. For this condition, we are looking for a block, a table, and a particular OnTop relationship so we enter "^herbal:WorkingMemory <Block>", "^herbal:WorkingMemory <Table>", and "^herbal:WorkingMemory <OnTop>" here. Of course you can match more than just Herbal working memory objects here. For example, matches based on the i/o link can also be placed in the "herbal:StateMatches"attribute.

Once you match objects on the top state, you will need to specify more detail about these objects using the "herbal:Exp" attribute. The "herbal:Exp" attribute allows us to specify more detail about the objects listed in the "herbal:StateMatches" attribute. Notice that this is done using Soar matching syntax. In this case we specify that the block and the table must be clear, and that the OnTop relationship must have the block on top, and cannot have the table on the bottom. Finally, we need to remember the current bottom block matched because we are going to have to mark this block as

"covered" when this operator is applied. This is done by creating a new match variable, "CurrentBottomBlock", that will be associated with OnTop relationship's current bottom block.

The final step to creating a condition is to specify what match variables should be made available to the operator that is using this condition, and this is done using the "Herbal:OperatorMemory" attribute. This makes it possible to later alter any of the objects referenced by these variables. When we create the action in the next step – the action responsible for actually moving the block to the table – we will see how the "Herbal:OperatorMemory" attribute is used. For now you can just specify that the block, table, OnTop relationship, and the current bottom block should be "remembered" after this condition matches the current situation.

Adding this condition to the MoveBlockToTable operator will result in the creation of the Soar operator proposal shown in Listing 2. Notice that the objects specified in the "Herbal:OperatorMemory" attribute are attached to the operator when it is proposed.

```
sp {propose*MoveBlockToTable
    (state <s>
        ^herbal:Name |BlocksWorldState|
        ^herbal:WorkingMemory <Block>
        ^herbal:WorkingMemory <OnTop>
        ^herbal:WorkingMemory <Table>)
     (<Block>
      ^blocks:Clear |true|
      ^blocks:Type |Block|)
     (<Table>
      ^blocks:Clear |true|
      ^blocks:Type |Table|)
     (<OnTop>
      ^blocks:Bottom <> <Table>
      ^blocks:Top <Block>
      ^blocks:Bottom <CurrentBottomBlock>)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name |MoveBlockToTable|
        ^Block <Block>
        ^CurrentBottomBlock <CurrentBottomBlock>
        ^OnTop <OnTop>
        ^Table <Table>)  }
```

Listing 2. Operator proposal generated based on the ClearBlockClearTable condition.

5. An operator gets its functionality from its conditions and actions. In the last step a condition was assigned to the "MoveBlockToTable" operator, and this condition resulted in the creation of an operator proposal in the generated Soar code. In the next few steps we will create and assign an action, and this action will result in the creation of an operator application in Soar.

   To create the action, click on the create next to the "Herbal:Actions" attribute of the "MoveBlockToTable" operator. This will display the new action dialog box.

6. When the new action dialog box is displayed you should assign the new action the attributes shown in Figure 19.

Protégé Tip: *Once again, when filling in the attributes you may need to arrange the forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

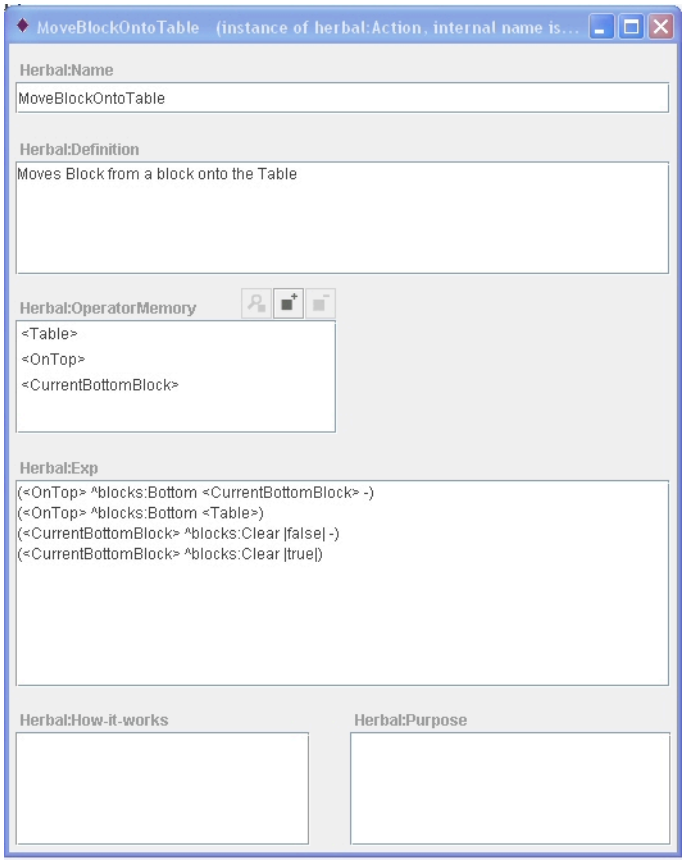

Figure 19. MoveBlockOntoTable action.

To better understand how an action object works, the attributes in this dialog box should be explained in more detail. Once again, the "herbal:Name" and "herbal:Definition" are self explanatory, so we should start the "herbal:OperatorMemory" attribute. The "herbal:OperatorMemory" object makes it possible for the modeler to specify what objects it will be using to perform its action. These objects **must** exist in the union of all of the "herbal:OperatorMemory" attributes of the conditions associated with this operator. In other words, you cannot manipulate an object in an action unless it has been matched and remembered by one of the conditions used by this operator.

In this case, the new action must have access to the current bottom block, the currently matched table, and the OnTop relationship for the block that is going to be moved to the table. Because the "ClearBlockClearTable" condition that we just created has

included these objects in its "Herbal:OperatorMemory" attribute, they will be available when this action is applied. This is the mechanism for pairing conditions and actions.

The most important attribute of an "Herbal:Action" object is the "Herbal:Exp". This is where the action is specified. In this case, the objective is to move the matched block on top of the matched table. This can be done by setting the OnTop relationship's bottom block equal to the table, and by clearing the previous bottom block's flag because it is no longer covered. Again, notice that Soar syntax is used to specify these actions. In addition, notice that namespace prefixes are used to specify attributes. For example, a block's Clear attribute is referenced using the model's namespace prefix ("blocks") that was assigned to the "BlocksTutorial" project when we originally created it with Protégé. The same is true for any other namespaces that have been imported into the project. For example, if we wanted to match on the Herbal name attribute we would use "herbal:Name".

The "MoveBlockOntoTable" action will result in the generation of the Soar operator application shown in Listing 3.

```
sp {apply*MoveBlockToTable
    (state <s>
        ^herbal:Name |BlocksWorldState|
        ^operator <o>)
    (<o> ^name |MoveBlockToTable|
        ^CurrentBottomBlock <CurrentBottomBlock>
        ^OnTop <OnTop>
        ^Table <Table>)
   -->
    (<OnTop> ^blocks:Bottom <CurrentBottomBlock> -)
    (<OnTop> ^blocks:Bottom <Table>)
    (<CurrentBottomBlock> ^blocks:Clear |false| -)
    (<CurrentBottomBlock> ^blocks:Clear |true|)  }
```

Listing 3. Operator application rule generated based on the MoveBlockOntoTable action.

7. The creation of the ClearBlockClearTable" condition and the "MoveBlockOntoTable" action complete the first Herbal operator. The completed "MoveBlockToTable" operator is shown in Figure 20.

Protégé Tip: *Once again, when filling in the attributes you may need to arrange the forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

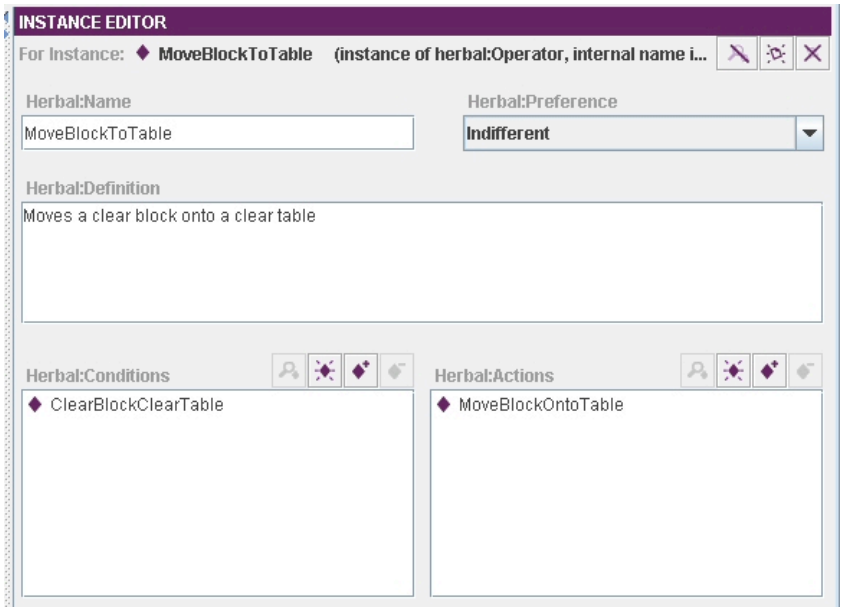

Figure 20. Completed MoveBlockToTable operator.

From this dialog box it is easy to tell what this operator does: it is proposed when there is a clear block and a clear table, and if applied it moves the block onto the table. In addition, this operator can be reused in other models or in a single model across multiple states. Finally, individual conditions and actions can also be easily reused by many operators or elaborations within a model, or in other models.

8. Once the MoveBlockToTable operator is complete, it should be assigned to the "Herbal:Operators" attribute of the top state. **Don't forget this step or else the operator proposal and operator application rules will not be generated.**

4.9 Creating the MoveBlockToBlock Operator and Assigning it to the Top State

1. The steps for creating the "MoveBlockToBlock" operator are similar to the steps used to create the "MoveBlockToTable" operator in the previous section. As a result, the completed operator, condition, and action dialog boxes are shown in Figures 21-23, and the generated operator proposal and operator application productions are shown in Listing 4. It is left to the reader to complete these dialogs and assign the operator to the top state.

Protégé Tip: *Once again, when filling in the attributes you may need to arrange the forms in Protégé using the Protégé Forms tab. Don't forget, you may also have to change the data entry control type to TextAreaWidget for fields that allow for multi-line text entry*

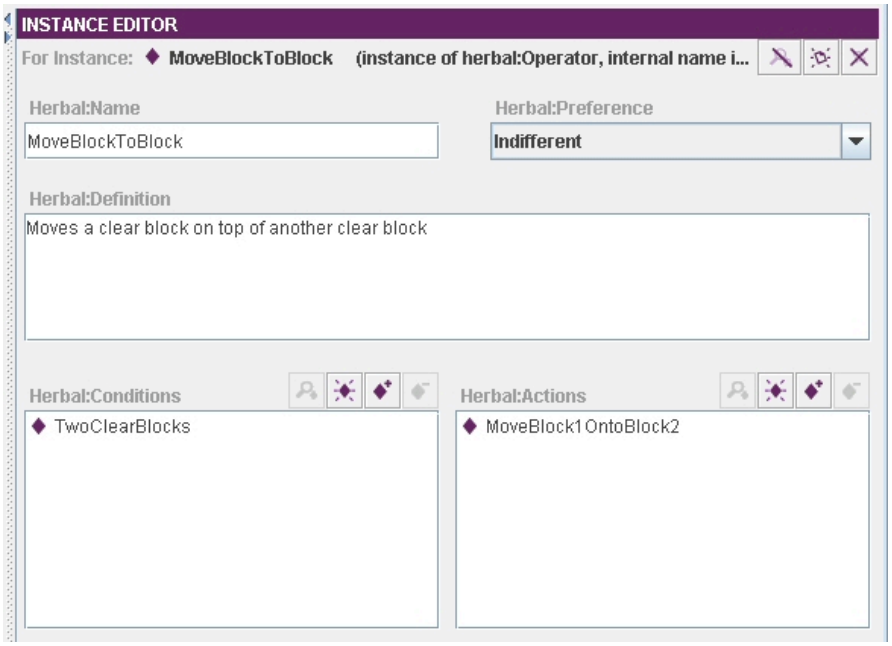

Figure 21. The completed MoveBlockToBlock operator.

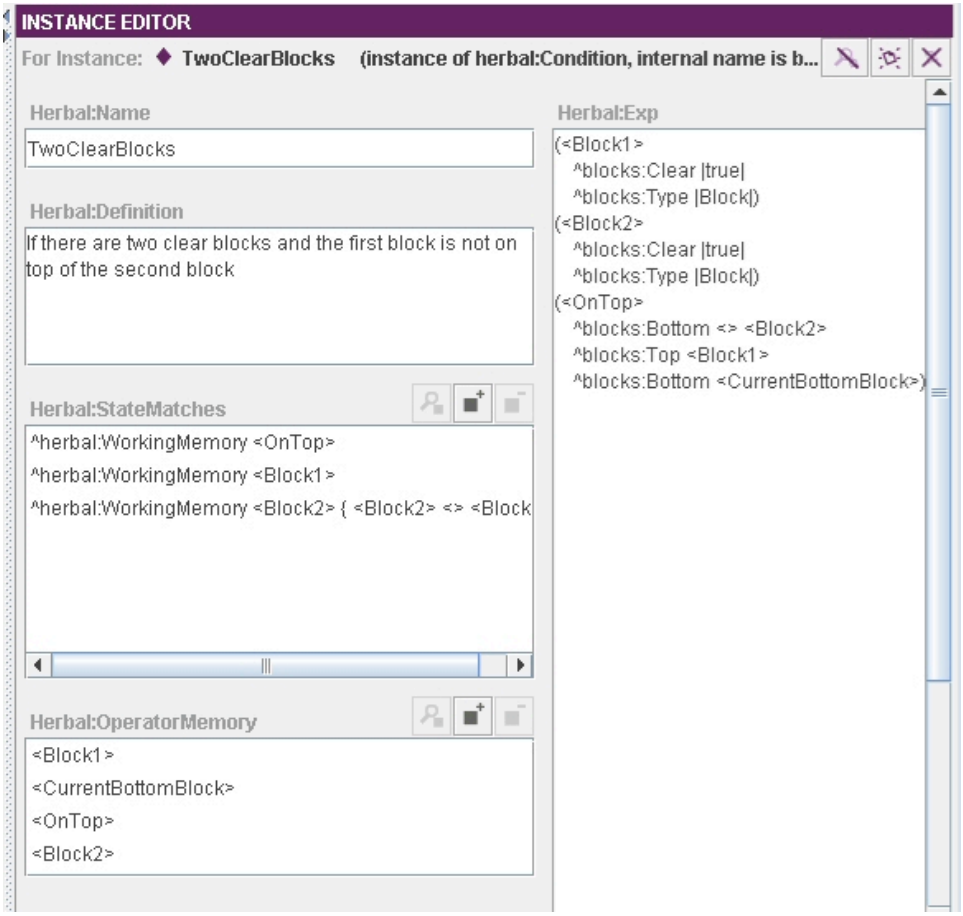

Figure 22. The complete TwoClearBlocks condition.

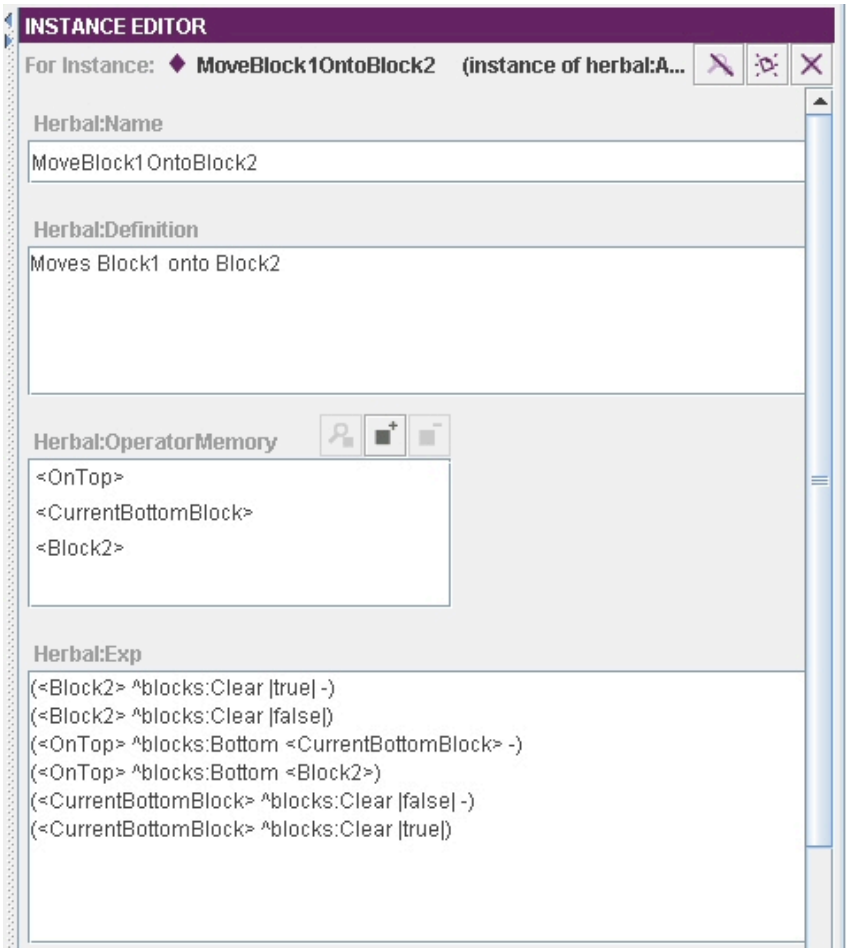

Figure 23. The complete MoveBlock1OntoBlock2 action.

```
sp {propose*MoveBlockToBlock
    (state <s>
        ^herbal:Name |BlocksWorldState|
        ^herbal:WorkingMemory <Block1>
        ^herbal:WorkingMemory <Block2> { <Block2> <> <Block1> }
        ^herbal:WorkingMemory <OnTop>)
    (<Block1>
    ^blocks:Clear |true|
    ^blocks:Type |Block|)
    (<Block2>
    ^blocks:Clear |true|
    ^blocks:Type |Block|)
    (<OnTop>
    ^blocks:Bottom <> <Block2>
    ^blocks:Top <Block1>
    ^blocks:Bottom <CurrentBottomBlock>)
   -->
    (<s> ^operator <o> + =)
    (<o> ^name |MoveBlockToBlock|
        ^Block1 <Block1>
        ^Block2 <Block2>
        ^CurrentBottomBlock <CurrentBottomBlock>
        ^OnTop <OnTop>)   }

sp {apply*MoveBlockToBlock
    (state <s>
        ^herbal:Name |BlocksWorldState|
        ^operator <o>)
    (<o> ^name |MoveBlockToBlock|
        ^Block2 <Block2>
        ^CurrentBottomBlock <CurrentBottomBlock>
        ^OnTop <OnTop>)
   -->
    (<Block2> ^blocks:Clear |true| -)
    (<Block2> ^blocks:Clear |false|)
    (<OnTop> ^blocks:Bottom <CurrentBottomBlock> -)
    (<OnTop> ^blocks:Bottom <Block2>)
    (<CurrentBottomBlock> ^blocks:Clear |false| -)
    (<CurrentBottomBlock> ^blocks:Clear |true|)  }
```

Listing 4. Operator proposal and operator application productions for the MoveBlockToBlock Operator

4.10 Creating the GoalStateReached Elaboration and Assigning it to the Top State

The final step in the model creation is to create an elaboration and assign it to the top state. This elaboration's responsibility will be to watch for the goal state, and when reached, terminate the model. Creating an elaboration is very similar to the creating an operator.

Elaborations have conditions and actions, and in fact, conditions and actions can be shared by and across operators and elaborations. To create the "GoalStateReached" elaboration follow these steps:

1. Make sure the "Instances" tab is selected and expand the "herbal:HerbalClass" node, and then the "herbal:Production": node in the instances tree.

2. Create a new "herbal:Elaboration" object and assign the following attributes:

Herbal:Name: GoalStateReached
Herbal:Definition: Terminates the simulation if block A is on top of block B, and block B is on top of block C, and block C is on top of the table.

3. Again, somewhat like operators, elaborations have conditions and actions. The "GoalStateReached" elaboration will have a single condition "IsGoalReached" and a single action "Done". These conditions and actions are created in the same fashion as the conditions and actions we already created for the operators, aside from one exception: the "Herbal:Operator" memory attributes are ignored for conditions and actions that are assigned to an elaboration. This is because elaborations are not proposed like operators but are immediately applied instead (this is reflected in that the chane in Soar are either O- or I- supported). As a result, it is not necessary to specify what matched objects will be "remembered". The completed "IsGoalReached" condition and "Done" action are shown in Figures 24 and 25.

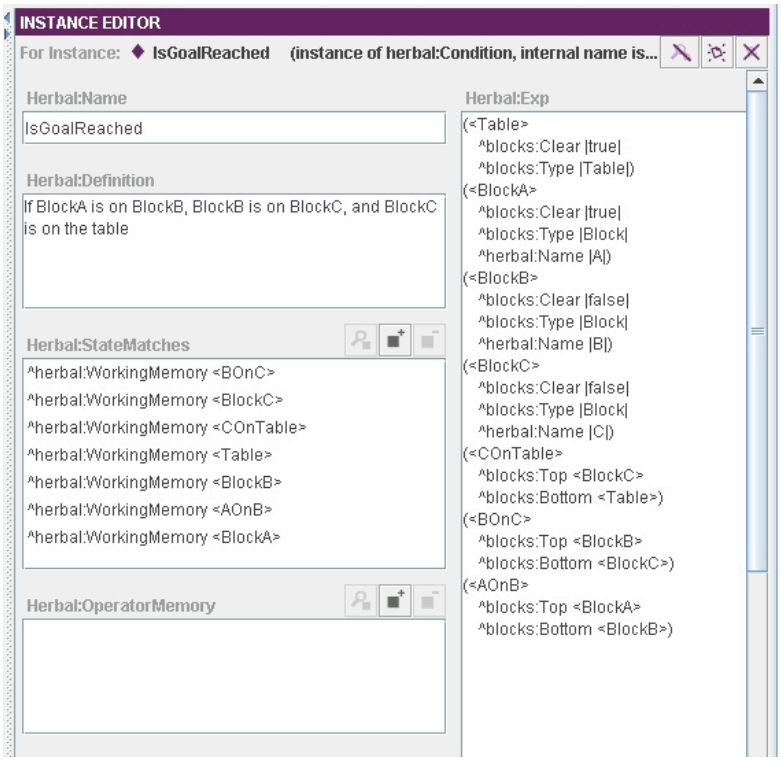

Figure 24. The complete IsGoalReached condition.

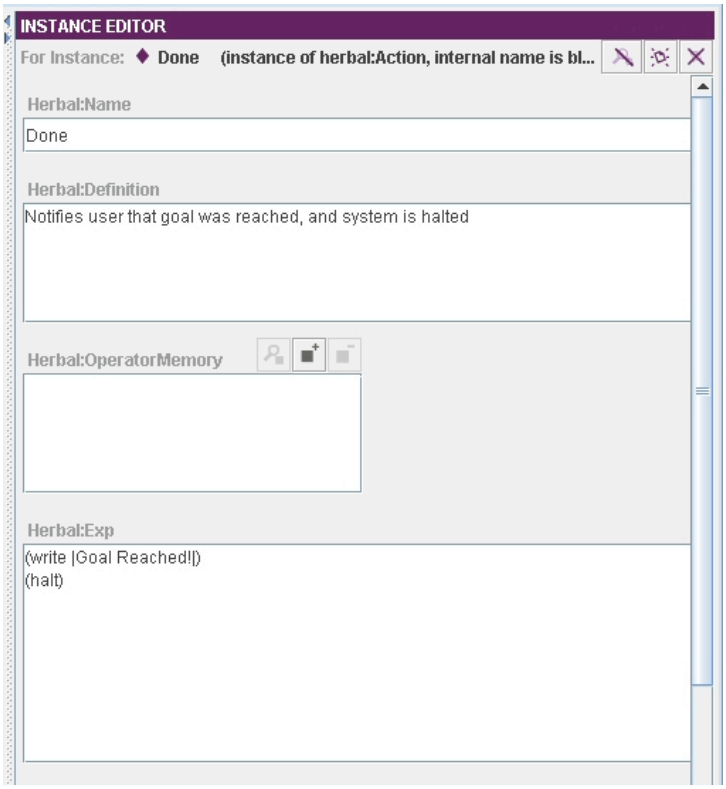

Figure 25. The Done action.

4. **Finally, don't forget to assign the "IsGoalReached" condition and "Done" action to the "GoalStateReached" elaboration, and then assign the "GoalStateReached" elaboration to the top state.** The completed "GoalStateReached" elaboration is shown in Figure 26 and the generated elaboration production is shown in Listing 5.

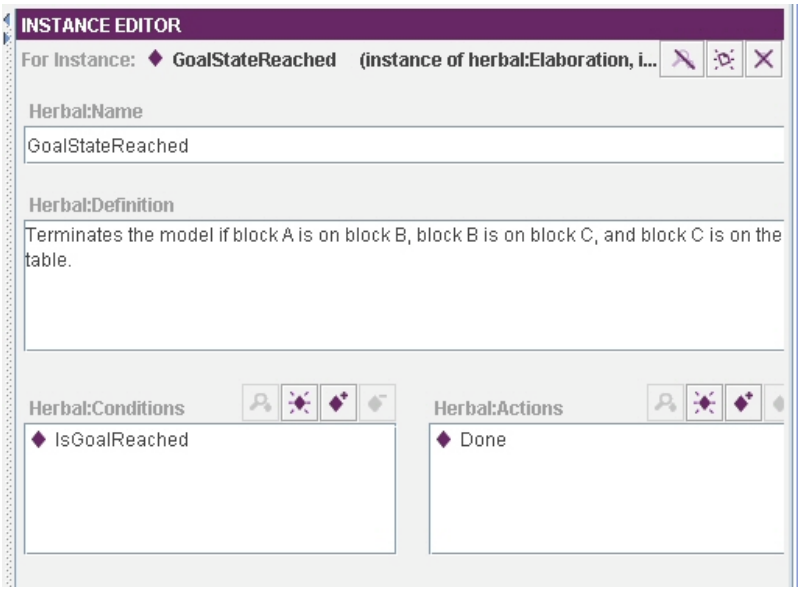

Figure 26. The completed GoalStateReached Elaboration.

```
sp {elaborate*GoalStateReached
    (state <s>
        ^herbal:Name |BlocksWorldState|
        ^herbal:WorkingMemory <AOnB>
        ^herbal:WorkingMemory <BOnC>
        ^herbal:WorkingMemory <BlockA>
        ^herbal:WorkingMemory <BlockB>
        ^herbal:WorkingMemory <BlockC>
        ^herbal:WorkingMemory <COnTable>
        ^herbal:WorkingMemory <Table>)
    (<Table>
    ^blocks:Clear |true|
    ^blocks:Type |Table|)
    (<BlockA>
    ^blocks:Clear |true|
    ^blocks:Type |Block|
    ^herbal:Name |A|)
    (<BlockB>
    ^blocks:Clear |false|
    ^blocks:Type |Block|
    ^herbal:Name |B|)
    (<BlockC>
    ^blocks:Clear |false|
    ^blocks:Type |Block|
    ^herbal:Name |C|)
    (<COnTable>
    ^blocks:Top <BlockC>
    ^blocks:Bottom <Table>)
    (<BOnC>
    ^blocks:Top <BlockB>
    ^blocks:Bottom <BlockC>)
    (<AOnB>
    ^blocks:Top <BlockA>
    ^blocks:Bottom <BlockB>)
   -->
    (write |Goal Reached!|)
    (halt)   }
```

Listing 5. The generated Soar code for the GoalStateReached elaboration.

5.0 Compiling the Blocks World Model

Once the model has been specified in Protégé, it is possible to compile the model into Soar productions. The automatic generation of Soar productions provided by the compiler helps in many ways including the reuse of Herbal objects such as operators, actions, and conditions. This reuse makes it possible to create the object once and use it multiple times. If you find an error and need to change the Herbal object, you only have to change it in one place and the compiler takes care of the rest.

To run the compiler from the "Model Attributes" tab, follow these steps:

1. Configure Protégé to use the "Herbal Compiler" tab. From the Protégé Project menu select the "Configure…" menu item. This will display the Configure dialog box. In this dialog box select the "Tab Widgets" tab and then check the "CompilerTab" check box. If you do not see "CompilerTab" listed here, please close down Protégé, make sure the "herbalWidgets.jar" file is located in the Protégé "plugins" folder, and then restart Protégé. This step only needs to be performed once per Herbal model.

2. Save the current version of your Blocks World Protégé project. Do not forget this step or your most recent changes will not be included in the compiled code!

3. Click on the "Herbal Compiler" tab located in the main Protégé window (see Figure 27).

4. Click on the "Generate Soar Code" button to launch the compiler (see Figure 27). When the compiler is done, a file named "model.soar" (containing Soar productions) will be created in the BlocksTutorial folder and also displayed in a window. After checking the code visually, you can close the window and then load the model into Soar for testing.

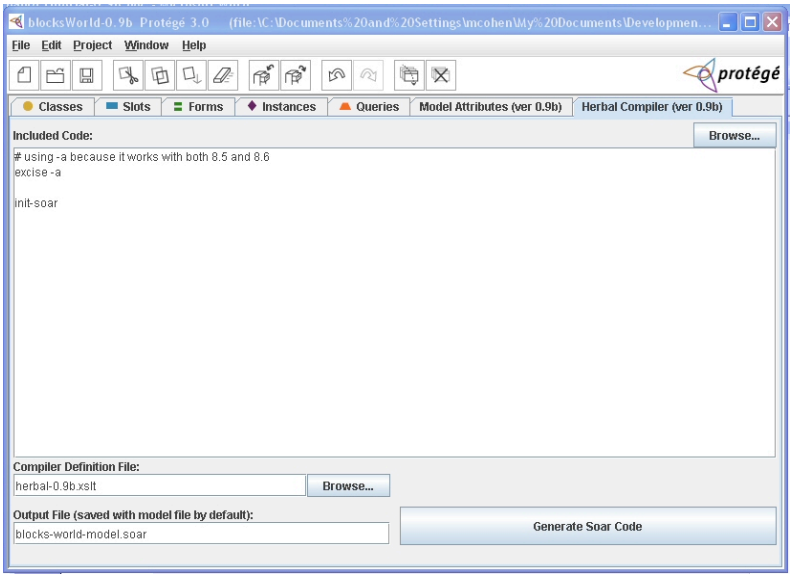

Figure 27. The Herbal Compiler Tab.

Operators Missing? *Are some of your operators or elaborations missing from your generated code? A common mistake is to forget to assign your operators and elaborations to the top state. This results in the operators never getting generated in the Soar code. If you can't find your operators in your code, go back and make sure they have been added to your top state.*

Recent Changes Missing? *Are your most recent changes missing from the generated code? A common mistake is to forget to save you model before clicking on the "Generate Soar Code: button. Try saving the model and regenerating the code.*

5.1 Including Custom Code in Your Model

It is possible to have Herbal include custom Soar rules or Tcl/Tk scripts automatically in the generated model. The Herbal Compiler tab contains a field named "Included Code" as shown in Figure 27. Any text entered into this field will be automatically inserted at the top of the code generated by Herbal. Text can be inserted into the "Included Code" field by either

manually typing the text in, or by inserting the contents of a file using the Browse button. If enter or insert any code in the "Included Code" field, be sure to save your model before running the compiler.

6.0 Impasses and Child States

Up to this point, the blocks world model you have built contains a single top level state named BlocksWorldState. However, it is often useful to use a hierarchy of states in order to add structure to a problem solving algorithm. This is best illustrated with an example.

In the current implementation of the blocks world model, it is possible for more than one MoveBlockToBlock operator to be valid. For example, in the initial state of the blocks world there are six different MoveBlockToBlock operators: move block A onto block B; move block A onto block C; move block B onto block A; move block B onto block C; move block C onto block A; and move block C onto block B.

Because the MoveBlockToBlock operator is marked as indifferent, the architecture will select one of the six operators for you. However, it may be desirable to include some problem solving logic that prefers one of the moves. For example, if we could prefer moving B onto C and then A onto B we could solve the problem much quicker. To accomplish this we need to generate an impasse that will create a child state. By adding operators to this new child state we can determine what operator is best and create a preference for that operator in the top state – thereby resolving the impasse.

By changing the preference for the MoveBlockToBlock operator from indifferent to acceptable, the model will no longer know how to choose the correct MoveBlockToBlock operator, and an Operator-Tie impasse will occur. This impasse will result in the creation of a new state, and from within this new state, new operators can help resolve the tie and choose the best MoveBlockToBlock operator.

6.1 Generating an Operator-Tie Impasse

The first step to utilizing hierarchical states for problem solving is to allow an impasse to occur. This can be done using the current blocks world model by changing the Herbal:Preference for the MoveBlockToBlock operator from indifferent to acceptable. In doing so, the model will encounter an operator-tie impasse whenever there is more than one MoveBlockToBlock operator to choose from.

Try changing the preference setting for the MoveBlockToBlock operator, recompiling the model, and running it in Soar. You should notice that the model immediately encounters an operator-tie impasse.

6.2 Creating an Herbal Child State

It is possible to create a child state in Herbal that will represent the state that is be generated by the impasse we just introduced in the previous section. You can assign operators to this

new child state that will perform actions to resolve the impasse and allow the top state to select the appropriate operator.

In Herbal, all child states are instances of the herbal:State class. To handle the operator-tie impasse we can create a new child state named ResolveTie. This can be done from the Protégé instances tab. Just highlight the herbal:State class, click on the create button, and set the new state's name attribute to "ResolveTie". We will return to this state later in order to add operators that will select the appropriate operator.

Now that the child state created has been created in Herbal, we need to create an Herbal impasse object that will create this child state when the impasse occurs.

6.3 Creating an Herbal Impasse

Creating an Herbal impasse that will handle the operator tie can be accomplished from the Protégé instances tab. To create the impasse, highlight the herbal:Impasse class and click on the create button. Once the impasse is created, attributes need to be set that name the impasse, specify its type, and designate the child state that will be created when the impasse occurs. In this case, the child state we want to be created is the one built in the previous section.

The following is a summary of the required attributes for the new herbal:Impasse object:

- herbal:Name = "Tie"
- herbal:ImpasseType = Operator-Tie
- herbal:SubState = ResolveTie

Finally, this new "Tie" impasse needs to be associated with the state that we expect the impasse to occur. In this case, this is the top state because this is the state in which the operator tie will take place.

This association can be accomplished by adding the Tie impasse to the Herbal:Impasses attribute of the BlocksWorldState.

6.4 Creating Operators that Resolve the Tie

With the child state created, and an impasse that will generate this new child state, we are ready to add the operators that will resolve the tie and return operation to the top state; as was accomplished much earlier in the tutorial using operators. Therefore, new operators need to be created and added to the ResolveTie state created in section 6.2.

One possible implementation is to create two new operators: one that generates a preference in the top state to place A on top of B; and another operator that generates a preference in the top state to place B on top of C. Because it is better to place B on top of C before A is placed on B, a second operator should be given a "best" preference and the later operator an "indifferent" preference (other combinations of preferences are possible that will accomplish the same result).

Creating these operators and adding them to the ResolveTie operator involves the same skills introduced earlier in the tutorial. As a result, you should already familiar with how to accomplish these tasks. As an exercise, you are encouraged to try this on your own. However, for your reference these two operators are included in the Block World Model that is part of the Herbal distribution. If you need help, you are encouraged to browse this model to see how these operators were created, and then attempt to reproduce these operators in your Blocks World Model.

7.0 The Herbal dTank Model

The Herbal distribution also contains a more interesting and complex model for use with the Java-based dTank game (acs.ist.psu.edu/dTank) created by the Applied Cognitive Science Lab at the Pennsylvania State University (Morgan, Ritter et al., 2005). Once you have completed this tutorial, and are familiar with the Herbal IDE and High-level Language, you may want to experiment with the dTank model.

The Herbal dTank model is located in the Herbal installation directory in a folder called "htank". Before you work with this model you should familiarize yourself with dTank by reading the dTank User's Manual (Isaac G. Council, Morgan, & Ritter, 2004). Once you are familiar with dTank you can view, edit, or compile the dTank Herbal model using the same skills learned in this tutorial.

The relative complexity of the dTank model – relative to Blocks World – makes it easier to see the advantages made possible by using the Herbal IDE. To illustrate these advantages, a brief summary of the structure of the dTank model is given in the following sections.

7.1 Summary of the dTank Model and How it Promotes Reuse

The Herbal dTank model consists of a single top state that contains eight operators and one elaboration. The elaboration named "remove-commands," is necessary in order to remove action commands that are placed on the Soar output link once they have been executed. All dTank models require this elaboration, and Herbal makes it very easy to reuse this elaboration across models.

The eight operators that are included in the Herbal dTank model are summarized in Table 9. These operators can be reused across models simply by importing the operators (and their components) into a new Herbal dTank model, or by creating a new Protégé project based on the "htank" project.

Table 9. A Summary of the Herbal dTank Operators

Name	Purpose
turn-left	If an enemy tank is spotted to my left, and my tank is not facing left, turn left
turn-right	If an enemy tank is spotted to my right, and my tank is not facing right, turn right
move-left	If an enemy tank is spotted to my left, and my tank is facing left, move forward one space
move-right	If an enemy tank is spotted to my right, and my tank is facing right, move forward one space
Wait	If there is no information to act on, wait until my sensors report something
aim-down	If there is not enemy tank seen, and my turret is facing up, rotate it down to look for an enemy
aim-up	If there is not enemy tank seen, and my turret is facing down, rotate it up to look for an enemy
Fire	If there is an enemy tank spotted on the same x-coordinate as my tank, fire!

In addition to the reuse of operators across different models, these operators also demonstrate code reuse within the same model. For example, nearly all of the operators listed in Table 9 share the same "basic-condition". This condition matches the input and output link and the current tank's status and visual field. Because this condition is reused by most of the operators in the model, a simple change to the "basic-condition" is immediately reflected in most of the operators in the model. Code regeneration imposes this change across several productions automatically, eliminating the need for copy and pasting and the possibility for typos. Table 10 lists all of the conditions used by the dTank model, along with the operators that use these conditions.

Table 10. A Summary of the conditions used in the Herbal dTank Model, and how these conditions are shared by operators.

Condition	Used By
Basic-condition	aim-up, aim-down, fire, move-left, move-right, turn-left, turn-right
commandIsComplete	Wait
facing-left	move-left
facing-right	move-right
no-tank-seen	aim-up, aim-down
not-facing-left	turn-left
not-facing-right	turn-right
nothingGoingOn	wait
tank-left	move-left, turn-left
tank-online	Fire
tank-right	move-right, turn-right
turret-down	aim-up
turret-up	aim-down

The Herbal dTank model also contains many of the common actions that are performed by a tank operating in the dTank environment. These actions are used by the Herbal model to operate the tank as a consequence of operator applications. Once again, the same type of "inter-model reuse" can be achieved by sharing these actions in the Herbal operators. A list of the actions included in the dTank model, along with the operators that share them, can be seen in Table 11.

Table 11. A summary of the actions used in the Herbal dTank Model, and how these actions are shared by operators.

Action	Used By
Fire	Fire
move-forward	move-left, move-right
removeCommand	remove-commands
rotate-turret-down	aim-down
rotate-turret-up	aim-up
setWait	Wait
Turn	turn-left, turn-right

7.2 A Summary of Herbal's Documentation Capabilities

In addition to promoting reuse, the Herbal Tank model also demonstrates Herbal's documentation capabilities. Every class in the Herbal High-Level Language Ontology contains explanation-oriented fields such as: definition, how-it-works, and purpose. These fields, filled in during model creation, are used by Herbal to generate useful documentation in the generated Soar productions. For example, as shown in Listing 6, the production that proposes that the tank move right contains a comment that was automatically generated by Herbal using the explanation oriented fields discussed above.

```
#-----------------------------------------------------------------------
# Production that proposes operator: move-right
# Operator Definition:
#   If the tank is to my right, and I am facing right, move right
# Proposal Conditions:
#   Basic condition set
#   true if I am facing right
#   true if a tank is to my right
#-----------------------------------------------------------------------
```

Listing 6. Herbal's automatically generated documentation.

7.3 How to Execute the Generated Herbal dTank Model

Executing the dTank productions generated by the Herbal dTank model is quite simple:

1. Run dTank as described in the dTank User's Guide (Isaac G. Council et al., 2004)

2. Run Soar as describe in the Soar User's Manual (Laird et al., 1999).

3. Load the Tcl/Tk script (soar-dTank_interface-2.0.7.tcl) located in the "htank" directory of the Herbal IDE installation folder (the command to load this script can also be included on the compiler tab as "included code" and therefore automatically inserted into the generated code for you).

4. Load the dTank Soar model generated by Herbal.

5. Run the model.

7.4 Suggestions for Expanding the Model

Once you have studied the Herbal dTank model and have executed it, you may want to expand the model in order to create a more intelligent tank. The following is just a short list of possible improvements. Hopefully this list will stir your imagination.

1. Create a new action that turns on the tank's shields and create new conditions that test to see if your tank is nearing death or under fire. Next, create an operator that utilizes your new action and conditions. For example, you could create an operator that turns on the tank's shields if it is near death or under fire.

2. Make the model more or less aggressive by modifying preferences on some of the operators.

3. Write a Tcl/Tk function that calculates the angle between two points and then utilize this function in a new operator that aims the turret towards an enemy (HINT: the basic tank that comes with the dTank distribution contains such a function).

4. Write an operator that recognizes nearby stones and remembers them by adding their location to working memory. You can then utilize this information to avoid or find stones later on.

8.0 Summary

The Herbal High-level Language and IDE are still being developed. As a result, you will find some bugs and some functionality missing. For example, currently the language does not support more than just a single, top-level state.

The time and effort you spend experimenting with Herbal is greatly appreciated and your feedback is very important. If you have any questions, or would like to report a bug, please contact Mark Cohen via email at mcohen@lhup.edu.

In addition, at this stage in development we might not be able to support backwards compatibility between models created with older versions of Herbal. As a result, if you create a model using the current version of Herbal, we ask that you do not expect your model to work with future releases of Herbal. It is possible that changes will be made to the Herbal environment that will break older models. At some point in time, as Herbal development stabilizes, an attempt will be made to maintain backward compatibility across future releases.

References

Cohen, A. M., Ritter, F. E., & Haynes, S. R. (2005). *Herbal: A high-level language and development environment for developing cognitive models in soar.* Paper presented at the 14th Behavior Representation in Modeling and Simulation (BRIMS), University City, CA.

Council, I. G., Haynes, S. R., & Ritter, F. E. (2003). *Explaining Soar: Analysis of Existing Tools and User Information Requirements.* Paper presented at the Proceeding of the International Conference on Cognitive Modeling, Bamberg, Germany.

Council, I. G., Morgan, G. P., & Ritter, F. E. (2004). *dTank: A Competitive Environment for Distributed Agents* (No. ACS 2004-1): Pennsylvania State Unviersity.

Haynes, S. R., Ritter, F. E., Council, I. G., & Cohen, M. A. (submitted). Explaining Intelligent Agents. *Manuscript submitted for publication*.

Laird, J. E., Congdon, C. B., & Coulter, K. J. (1999). *The Soar User's Manual Version 8.2*: University of Michigan.

Morgan, G. P., Cohen, A. M., Haynes, S. R., & Ritter, F. E. (2005). Increasing Efficiency of the Development of User Models. *IEEE System Information and Engineering Design Symposium*.

Morgan, G. P., Ritter, F. E., Cohen, M. A., Stevenson, W. E., & Schenck, I. N. (2005). *dTank: An Environment for Architectural Comparisons of Competitive Agents.* Paper presented at the 14th Conference on Behavior Representation in Modeling and Simulation (BRIMS), Universal City, CA.

Mozilla. (2004). *Mozilla Public License*. Retrieved June 15, 2004, 2004, from http://www.mozilla.org/MPL/MPL-1.1.html

Nuxoll, A., & Laird, J. E. (2003). *Soar Design Dogma*, from www.eecs.umich.edu/~soar/sitemaker/docs/misc/dogma.pdf

Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R., Gobet, F., & Baxter, G. D. (2003). *Techniques for modeling human and organization behavior in synthetic environments: A supplementary review*, Wright Patterson Air Force Base, OH: Human Systems Information Analysis Center.

Stanford Medical Informatics. (2003). *Protege 2000 User's Guide* (User's Guide): Stanford University.

Stanford Medical Informatics. (2004). Protege (Version 2.1.1).

W3C. (2004). *Resource Description Framework*, from www.w3.org/RDF/